Reg No: 223014851

Department of Computer and Software Engineering

Mobile Applications System and Design

27th February 2026

Topic: State Management in Flutter

Below are explanation of different concepts used in flutter application development

* Provider: is a simple state management solution in flutter used to share and update data across Widgets.
it uses ChangeNotifier to notify the UI when the state changes.

* Riverpod: is improved version of Provider, it does not depend on BuildContext and is more flexible and testable.

* Bloc (Business logic Component): Block is structured state management approach that separates business logic from UI.
It works using events and states and uses streams.

* GetX: is a light weight flutter framework for state management, routing and dependency injection.
GetX is known for fast development and minimal boiler plate code

key features: * GetX is known for very fast development
* GetX Provides High performance.
* Easy syntax.

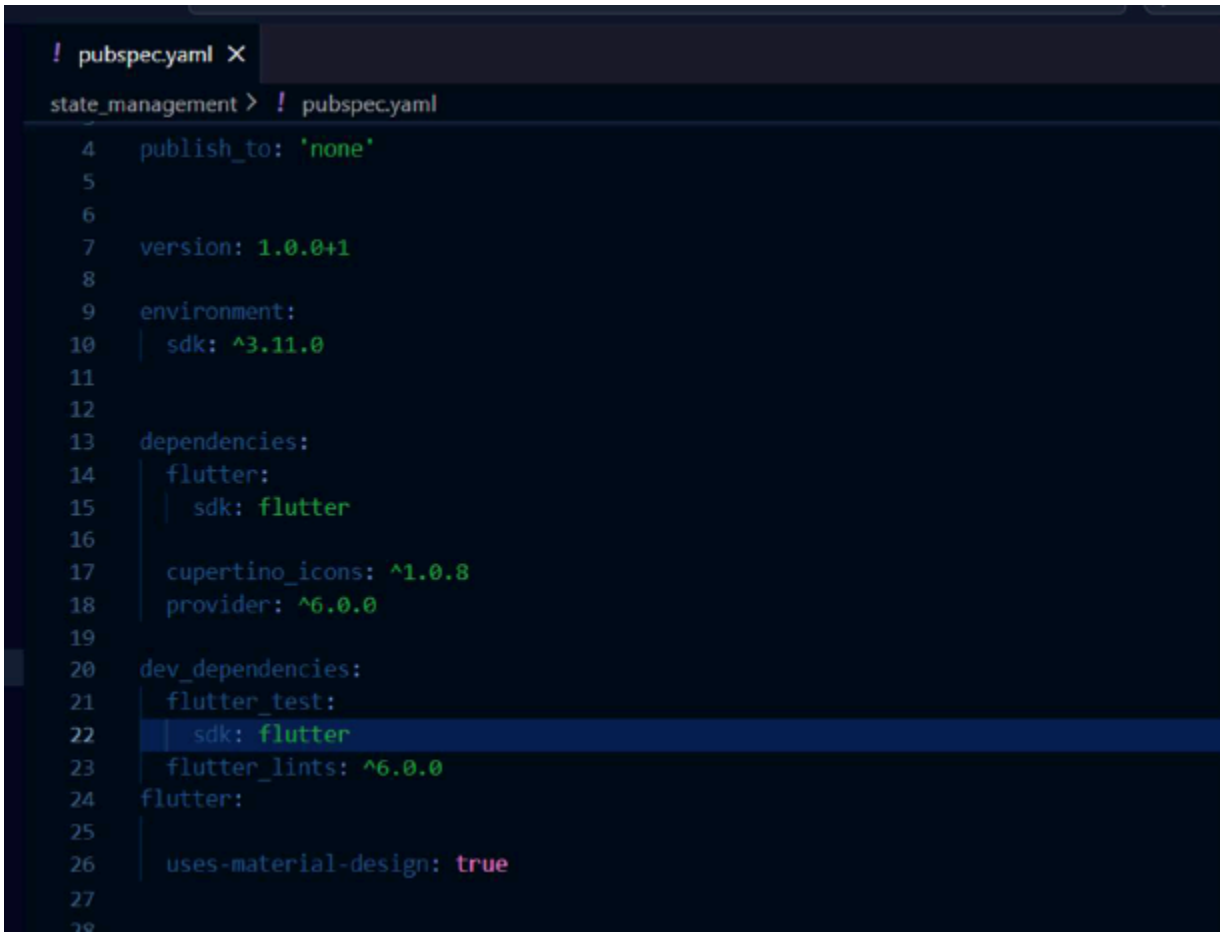
Q2) Table explaining where each state management is applicable

Situation	Provider	Riverpod	Bloc	GetX
Small Applications	Very good	Good	Too complex	Very good
Medium Applications	Good	Very good	Good	Very good
Large / Enterprise Applications	Limited	Good	excellent	Moderate
Team Projects	Moderate	Good	excellent	depends on team
Fast development	Good	Moderate	Slower	excellent
Strict Architecture Requirement	limited	Good	excellent	Limited

3. Implementation Steps

Step 1: Adding the Dependency

Dependencies: here you have to add **provider** dependency in **pubsec.yaml**



After adding the dependency:

Run

Explanation

The `provider` package is added to enable state management functionality.
Running `flutter pub get` downloads and installs the package into the project.

This step prepares the project to use Provider features.

Step 2: Creating the State Class

```dart
import 'package:flutter/material.dart';

class CounterProvider extends ChangeNotifier {
 int _count = 0;

 int get count => _count;

 void increment() {
   _count++;
   notifyListeners();
 }

 void decrement() {
   _count--;
   notifyListeners();
 }
}
```

Explanation

- `CounterProvider` manages the application state.
- It extends `ChangeNotifier`, allowing it to notify widgets when changes occur.
- `_count` stores the counter value.
- `increment()` and `decrement()` modify the state.
  `notifyListeners()` triggers UI updates.

This class separates business logic from the user interface.

Step 3: Providing the State to the App

```
void main() {
 runApp(
   ChangeNotifierProvider(
     create: (_) => CounterProvider(),
     child: MyApp(),
   ),
 );
}
```

Explanation

- `ChangeNotifierProvider` makes the state accessible throughout the app.
- `create` initializes the provider.
- All widgets inside `MyApp` can now access `CounterProvider`.

This step connects the state to the entire application.

Step 4: Building the Application UI

```
class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     home: CounterScreen(),
   );
 }
}
```

Explanation

- `MyApp` defines the root structure.
- `MaterialApp` sets up the Flutter app.
- `CounterScreen` becomes the main screen.

This defines the overall application layout.

Step 5: Accessing the State (Listening)

```
Consumer<CounterProvider>(
 builder: (context, counter, child) {
   return Text(
     counter.count.toString(),
     style: TextStyle(fontSize: 40),
   );
 },
);
```

Explanation

- `Consumer` listens to `r`.
- Whenever `notifyListeners()` is called, this widget rebuilds.
- `counter.count` retrieves the latest value.

This ensures the UI reflects the current state automatically.

Step 6: Updating the State

```
FloatingActionButton(
 onPressed: () {
   Provider.of<CounterProvider>(context, listen: false)
       .increment();
 },
 child: Icon(Icons.add),
);
```

Explanation

- When the button is pressed, `increment()` is executed.
- `listen: false` prevents unnecessary rebuilding.
  The state changes.
  `notifyListeners()` triggers UI update.

This makes the application reactive and dynamic.

Flow Summary

1. The provider is initialized at the top level.
2. The UI listens using `Consumer`.
3. The user presses the button.
4. The counter value changes.
5. `notifyListeners()` runs.
6. The UI rebuilds automatically.

This demonstrates clean separation of logic and presentation.

Output

The application displays:

- A counter number in the center.
- A "+" button to increase the value.
- A "−" button to decrease the value.
- The number updates instantly when buttons are pressed.

Conclusion

This lab demonstrates how Provider simplifies state management in Flutter by separating business logic from UI and automatically rebuilding widgets when state changes. Provider is efficient, scalable, and ideal for small to medium-sized applications.